31.8.–3.9.2015
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Aus, Sitz, Platz!

CDI-Komponenten lokal ohne Server testen

Werner Eberling
MATHEMA Software GmbH

Email: werner.eberling@mathema.de
Twitter: @Wer_Eb

Kurze Vorstellung

- Sprecher

Werner Eberling

Principal Consultant / Autor

Email: werner.eberling@mathema.de

Twitter: @Wer_Eb

Das Problem

- „Ich kann keine vernünftigen Unit-Tests schreiben, ohne ständig den Application Server hoch und runter zu fahren.“

- CDI Komponenten laufen in einem Application Server
  - Lange „Round Trips“
  - Komponenten nutzen Container Ressourcen

Die Lösung

- CDI Container existieren auch für Java SE Umgebungen!
  - Weld
    - mvn-group-id: org.jboss.weld.se
    - mvn-artifact-id: weld-se

- Apache Deltaspike verhindert „vendor lock-in“

- Und was ist mit den Container Ressourcen?
  - Später… ,)

Apache Deltaspike – modules

- Herstellerunabhängige Erweiterungen
  - Kompatibel mit allen gängigen CDI Implementierungen
  - Unterschiedliche Projekte (aka. modules) zu verschieden Problemdomänen

- Module rund um das Thema Unit-Testing
  - Core
    - mvn-artifact-id: deltaspike-core-api
  - Container Control
    mvn-artifact-id: deltaspike-cdictrl-api
  - Test Control
    - mvn-artifact-ids: deltaspike-test-control-module-api, deltaspike-test-control-module-impl

Apache Deltaspike – Container Integration

- CDI Container sind NICHT Teil von Deltaspike
  - Lokale Container müssen explizit hinzugefügt werden
  - Test mit dem Container der Wahl möglich
- Integration für die gängigen Implementierungen vorhanden
  - Weld
    - mvn-artifact-id: deltaspike-cdictrl-weld
  - OpenWebBeans
    - mvn-artifact-id: deltaspike-cdictrl-owb

Unit-Tests mit Deltaspike – Test Struktur (1)

- CDI Container vor Durchführung der Tests starten (container control module)

```
CdiContainer cdiContainer =
CdiContainerLoader.getCdiContainer();
cdiContainer.boot();
cdiContainer.getContextControl().startContexts();
```

- Nachschlagen von Bean-Referenzen über die Klasse *BeanProvider* (core module)

```
Customer customer =
    BeanProvider.getContextualReference(Customer.class);
```

Unit-Tests mit Deltaspike – Test Struktur (2)

- Mit der Bean-Referenz die notwendigen Prüfungen durchführen

```
Assert.assertEquals("Doe", customer.getName() );
```

- Am Ende den Container wieder herunterfahren

```
cdiContainer.shutdown();
```

Unit-Tests mit Deltaspike – Best Practices

- Start des Containers kann zeitintensiv sein
  - Am besten nur einmal pro Testklasse (@BeforeClass)
  - Am Ende der Testklasse wieder stoppen (@AfterClass)

- Nicht vergessen für jede Testmethode die Kontexte neu zu starten!

```
cdiContainer.getContextControl().stopContext(RequestScoped.class);

cdiContainer.getContextControl().startContext(RequestScoped.class);
```

Unit-Tests mit Deltaspike – Injections

- CDI Beans können auch in die Testklasse injiziert werden

```
BeanManager beanManager = cdiContainer.getBeanManager();
CreationalContext creationalContext =

beanManager.createCreationalContext(null);
AnnotatedType annotatedType =
        beanManager.createAnnotatedType(this.getClass());
InjectionTarget injectionTarget =
        beanManager.createInjectionTarget(annotatedType);
injectionTarget.inject(this, creationalContext);
```

Unit-Tests mit Deltaspike – CdiTestRunner

- Mehr Komfort durch Deltaspikes *CdiTestRunner*
  - Kümmert sich um Starten/Stoppen des Containers
  - Automatische Dependency Injection

```
@RunWith(CdiTestRunner.class)
public class CustomerServiceTest {
 @Inject
 CustomerService customerService;

 // Testmethoden ...
}
```

Dummies statt Container Ressourcen (1)

- CDI Komponenten nutzen i.d.R. Ressourcen des Containers

```
@Named @ApplicationScoped
public class CustomerServiceImpl implements CustomerService {
    @Inject
    private Principal currentUser;

    //...
}
```

- Im lokalen Unit-Test existieren diese Ressourcen nicht

Dummies statt Container Ressourcen (2)

- Bereitstellung eines *Principal* Dummies über eine *Producer* Methode

```
public class PrincipalDummyProducer {
    @Produces @SessionScoped
    public Principal createDummyPrincipal() {
        return new PrincipalDummy();
    }
}
```

Mockups für Fremdservices (1)

- Eine CDI Komponente nutzt auch andere Komponenten (aka. Services)

```
@Named
@ApplicationScoped
public class CustomerServiceImpl implements CustomerService {
    @Inject
    private AuditService auditService;

    //...
}
```

- Ein Unit-Test soll aber nur die einzelne Komponente testen

Mockups für Fremdservices (2)

- Einführung eines Service-Mockups durch Spezialisierung

```
@Specializes
public class AuditServiceMockupImpl extends AuditServiceImpl {
    private Principal userReported;
    @Override
    public void logAction(Principal user, String action) {
        userReported = user;
    }
    // Additional method to ease testing
    public Principal getUserReported(){
        return userReported;
    }
}
```

Mockups für Fremdservices (3)

- Auch der Unit-Test kann den Mockup referenzieren, um Aufrufe zu prüfen

```
@RunWith(CdiTestRunner.class)
public class CustomerServiceTest {
    @Inject
    AuditServiceMockupImpl auditService;
    //...
    @Test
    public void testSaveIsLoggedCorrectly() {
        //...
        Assert.assertEquals(currentUser.getName(),
                            auditService.getUserReported().getName());
    }
}
```

Unit-Tests und Persistenz (1)

- Services laden und speichern Daten über JPA
  - *EntityManager* wird vom Server bereitgestellt
- Testbarkeit liegt im Design begründet
  - Nie *@PersistenceContext* direkt verwenden
- Auch in einer JEE Umgebung den CDI Weg nutzen
  - *@Inject*
  - Producer Komponente

Unit-Tests und Persistenz (2)

- EntityManager Producer für JEE Umgebungen (Beispiel)

```
@ApplicationScoped
public class EntityManagerProducer {
    @PersistenceContext
    private EntityManager em;
    @Produces @RequestScoped
    public EntityManager getEntityManager(){
        return em;
    }
}
```

Unit-Tests und Persistenz (3)

- Durch Spezialisierung den lokalen EntityManager einführen

```
@Specializes @ApplicationScoped
public class LocalEntityManagerProducer extends EntityManagerProducer{
    private EntityManagerFactory emf;
    // create emf in @PostConstruct and close it in @PreDestroy...
    @Produces @RequestScoped
    public EntityManager getEntityManager() {
        return emf.createEntityManager();
    }
    public void closeEntityManager(@Disposes EntityManager em){
        em.close();
    }
}
```

Unit-Tests und Persistenz (4)

- Wo werden die Daten gespeichert?
  - In lokalen oder In-memory Datenbanken (e.g. Derby, H2, …)

```
<persistence-unit name="customerDB"
                  transaction-type="RESOURCE_LOCAL">
    <properties>
      <property name="javax.persistence.jdbc.url"
                value="jdbc:derby:memory:myDB;create=true"/>
      <property name="javax.persistence.jdbc.driver"
                value="org.apache.derby.jdbc.EmbeddedDriver"/>
      <property name="hibernate.hbm2ddl.auto"
                value="create"/>
    </properties>
</persistence-unit>
```

Unit-Tests und Persistenz (5)

- Wer kümmert sich um die Transaktionen?
- Qual der Wahl
  - Definition eines Interceptor Bindings und Implementierung eines TX-Interceptors
    - JEE-Szenario: Delegation an TX-Manager
    - Lokales Szenario: Nutzung der EntityTransaction
  - Oder einfach das Deltaspike JPA Module verwenden

31.8.–3.9.2015
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Vielen Dank!

Werner Eberling

MATHEMA Software GmbH

Email: werner.eberling@mathema.de

Twitter: @Wer_Eb